

Satisfaction of a Quantified Predicates

- We haven't discussed how to decide whether a state satisfies a quantified predicate. If the state does not contain the quantified variable, it is not hard to understand the problem: does $\{y = 1\}$ satisfies $\forall x. x^2 \geq y - 1$? But what if the quantified variable is in the state: does $\{z = 4, x = -5\} \models \exists x. x \geq z$?
 - $\sigma \models \exists x \in S. p$ if for one or more **witness values** $\alpha \in S$, it's the case that $\sigma[x \mapsto \alpha] \models p$.
- True or False?
 - $\{z = 4, x = -5\} \models \exists x. x \geq z$?
True. We can find $x = 5$ such that $\{z = 4, x = -5\}[x \mapsto 5] = \{z = 4, x = 5\}$ satisfies $x \geq z$.
 - $\sigma \models \exists x. x^2 \leq 0$?
True. x has domain \mathbb{Z} , and we can find $x = 0$ such that $\sigma[x \mapsto 0]$ satisfies $x^2 \leq 0$.
 - From these examples, we can see that if a variable is bounded in an existential quantifier, its current value in a state doesn't affect the satisfaction of the state.
 - Which of the following state satisfies $x < 3 \wedge \exists x. b[x] > 5$?
 - $\{x = 0, b = (2, 4, 3, 1)\}$
 - $\{x = 1, b = (1, 3, 5, 7)\}$
 - $\{x = 2, b = (1, 3, 5, 4)\}$
 - $\{x = 3, b = (6, 5, 3, 1)\}$
 - $\sigma \models \forall x \in S. p$ if for every value $\alpha \in S$, we have $\sigma[x \mapsto \alpha] \models p$.
- True or False.
 - $\{y = 1\} \models \forall x \in \mathbb{Z}. x^2 \geq y - 1$?
True. $\{y = 1\}(y - 1) = 0$, and we know that for all integer α , we have $\alpha^2 \geq 0$.
 - $\{x = -1\} \models \forall x \in \mathbb{Z}. x^2 \geq x$?
True. We know that for all integer α , we have $\alpha^2 \geq \alpha$.
 - From this example, we can see that if a variable is bounded in a universal quantifier, its current value in a state doesn't matter as well.
 - How about "doesn't satisfy"? Without consider the possible runtime error during the evaluation, we can use " $\sigma \not\models p \Leftrightarrow \sigma \models \neg p$ " then apply DeMorgan's Law here:
 - $\sigma \not\models \exists x \in S. p \Leftrightarrow \sigma \models \neg \exists x \in S. p \Leftrightarrow \sigma \models \forall x \in S. \neg p$
 - $\sigma \not\models \forall x \in S. p \Leftrightarrow \sigma \models \neg \forall x \in S. p \Leftrightarrow \sigma \models \exists x \in S. \neg p$

Validity of Predicates

- Let p be a proposition or predicate. $\models p$ means $\sigma \models p$ for all σ , and we say p is **valid**. In other words, we can say "It is always true that, p ".
 - $\models p \Leftrightarrow \forall \sigma \in S. \sigma \models p$ (where S is the collection of all well-formed states that are proper for p)
- $\not\models p$ means $\sigma \not\models p$ for **some** σ , and we say p is **invalid**. In other words, we can say " p is not always true".
 - $\not\models p \Leftrightarrow \exists \sigma \in S. \sigma \not\models p$ (where S is the collection of all well-formed states are proper for p)

4. True or False.

- a. $\models x > 1 \rightarrow x^2 > x$ True, because all states satisfy $x > 1 \rightarrow x^2 > x$
b. $\models x > 1 \wedge x^2 > x$ False, because $\{x = 0\} \not\models x > 1 \wedge x^2 \geq x$

- c. $\models \forall x. x > 1 \rightarrow x^2 > x$ True

We need to check that whether $\forall \sigma. \sigma \models \forall x. x > 1 \rightarrow x^2 > x$. While check whether some $\sigma \models \forall x. x > 1 \rightarrow x^2 > x$, we don't need any bindings in σ , since this universal quantified x and its body contains only variable x as well. Therefore, we only need to check whether any one $\sigma \models \forall x. x > 1 \rightarrow x^2 > x$, and we can simple pick $\sigma = \emptyset$ to process.

- If p is a predicate about x and x is not a variable being bounded in p , then $\models p \Leftrightarrow \models \forall x. p$.

- d. $\models \forall x. x > 1 \wedge x^2 > x$ False

- e. $\models \exists x. x > 1 \wedge x^2 > x$ True

5. Is the following predicate valid?

$$\exists y. y \neq 0 \wedge x * y \neq 0$$

$\exists y. y \neq 0 \wedge x * y \neq 0$ is a predicate about x and x is not the variable being bounded here, so semantically its validity is equivalent to $\models \forall x. \exists y. y \neq 0 \wedge x * y \neq 0$; then it is easy to see this predicate is invalid.

To show it is invalid, we can argue that:

$$\begin{aligned} &\not\models \exists y. y \neq 0 \wedge x * y \neq 0 \\ &\Leftrightarrow \exists \sigma. \sigma \not\models \exists y. y \neq 0 \wedge x * y \neq 0 && \text{definition of invalid} \\ &\Leftrightarrow \exists \sigma. \sigma \models \neg \exists y. y \neq 0 \wedge x * y \neq 0 && \text{definition of } \sigma \not\models p \\ &\Leftrightarrow \exists \sigma. \sigma \models \forall y. \neg(y \neq 0 \wedge x * y \neq 0) && \text{DeMorgan's Law} \\ &\Leftrightarrow \exists \sigma. \sigma \models \forall y. y = 0 \vee x * y = 0 && \text{DeMorgan's Law} \end{aligned}$$

We can find that $\sigma = \{x = 0\}$ is a witness, since for all possible values of y , we always have $y = 0$ or $x * y = 0$.

Syntax of Statements in Our Programming Language

- **Program:** A program is simply a statement, typically a sequence statement.
- In general, a statement is a standalone unit of execution whose purpose is not creating a value (opposite to expression) but creating changes in memory. We usually use letter S to represent a statement in our programming language.
- We initially introduce 5 types of statements here, and we will introduce more in future classes.
 - **No-op** statement: **skip**
It simply means do nothing.
 - **Assignment** statement: $v := e$ or $b[e_0][e_1] \dots [e_{n-1}] := e$
Assigning expression e to variable v or assigning expression e to a certain index in an n -dimensional array b .
 - **Sequence** statement: $S; S'$

Do S then do S' . Note that S' can be another sequence statement, then we have a longer sequence like: $S_1; S_2; S_3$.

- **Conditional statement: $\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}$**

Do S_1 if B is evaluated to *True*, do S_2 if B is evaluated to *False*.

- A conditional statement and a conditional expression can look alike, we tell one another by context. Note that S_1 and S_2 both must be statements.
- When S_2 is a no-op statement, then we can simplify it from **$\text{if } B \text{ then } S_1 \text{ else skip fi}$** to **$\text{if } B \text{ then } S_1 \text{ fi}$** so we don't need to formally define a **if – then** statement.

- **Iterative statement: $\text{while } B \text{ do } S \text{ od}$**

A “while loop” with loop condition B and do S in each iteration.

- We don't have “for loops” in our language but we can simulate it using **while – do**. For example, if we need: **$\text{for } x = e_1 \text{ to } e_2 \text{ do } S$** , we turn it into: **$x := e_1; \text{while } x < e_2 \text{ do } S; x := x + 1 \text{ od}$**

6. Translate the following Java statements into statements in our programming language.

a. $x = (y == 2) ? 5 + x : 6;$

$x := \text{if } y = 2 \text{ then } 5 + x \text{ else } 6 \text{ fi}$

b. **$\text{if } (y == 2) \{x = 5 + x;\} \text{ else } \{x = 6;\}$**

$\text{if } y = 2 \text{ then } x := 5 + x \text{ else } x := 6 \text{ fi}$

c. **$\text{if } (y == 2) \{x = 5 + x;\} x = 6$**

$\text{if } y = 2 \text{ then } x := 5 + x \text{ fi}; x := 6$ or **$\text{if } y = 2 \text{ then } x := 5 + x \text{ else skip fi}; x := 6$**

7. Create a program that calculates the power of 2. We run it with input integer n and returns $y = 2^n$; unless $n < 0$, in which case we return 0.

If we write it with indentation, then one way to write it is as follows.

```

if  $n < 0$  then
     $y := 0$ 
else
     $x := 0;$ 
     $y := 1;$ 
    while  $x < n$ 
    do
         $x := x + 1;$ 
         $y := y + y$ 
    od
fi

```

It is also acceptable to write it in one line:

$\text{if } n < 0 \text{ then } y := 0 \text{ else } x := 0; y := 1; \text{while } x < n \text{ do } x := x + 1; y := y + y \text{ od fi}$

8. Translate the following C statements into statements in our programming language.

a. $x = a * ++z$

$z := z + 1; x := a * z$

- b. $x = a * z + +$
 $x := a * z; z := z + 1$
- c. **while** ($--x \geq 0$) $z *= x$;
 $x := x - 1$; **while** $x \geq 0$ **do** $z := z * x; x := x - 1$ **od**
- d. **while** ($x -- \geq 0$) $z *= x$;
while $x \geq 0$ **do** $x := x - 1; z := z * x$ **od**; $x := x - 1$